
Java course - IAG0040
I/O, Files & Streams
Anton Keks
2010

File System

- **java.io.File** provides system-independent view of hierarchical pathnames (immutable)
  - `File f = new File("/bin");`
    `f = new File(f, "ping");`
- Can be used to represent files or directories
  - check for existence and permissions
  - query various info (length, attributes)
  - Create, rename or delete both files and directories
  - static fields provide quick access to system-dependent separators: `File.separator`, `File.pathSeparator`
  - `'/'` works on all platforms, including Windows

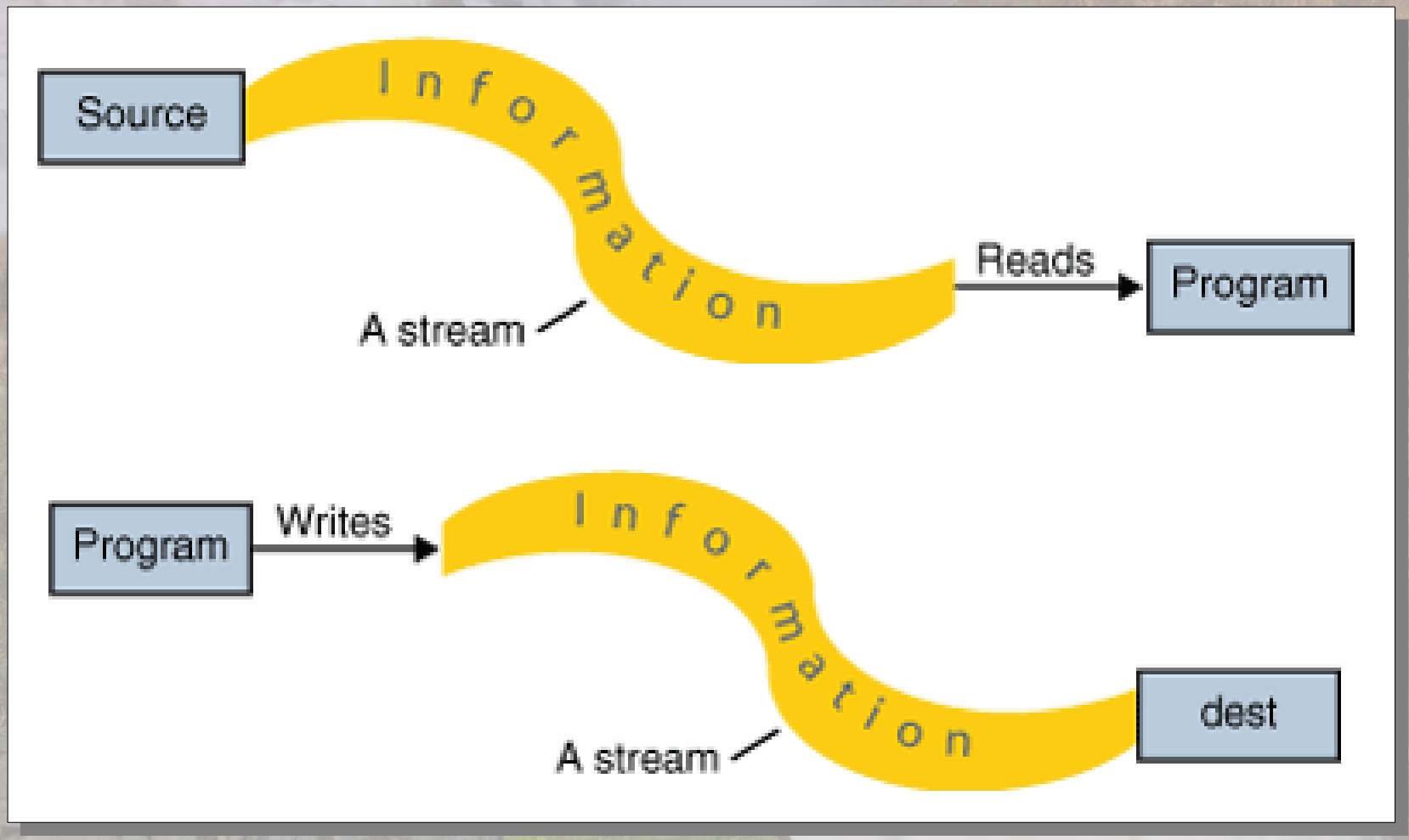
File System Tips

- How to avoid dealing with separators
  - `File parent = new File("someDir");`
  - `File subdir = new File(parent, "bin");`
- Obtain a valid temporary file
  - `File tmpFile = File.createTempFile("something", ".tmp");`
  - `tmpFile.deleteOnExit();`
- Enumerate Windows drives
  - `File[] roots = File.listRoots();`
  - `File unixRoot = roots[0];`
- Enumerate files in a directory
  - `File[] files = new File("someDir").listFiles();`

Random Access

- java.io.RandomAccessFile
  - is used when streams are not suitable, e.g. random access of large binary files
  - like a large array of bytes in the file system
  - both reading and writing is possible (depending on mode)
- Has a file pointer that can be read and moved
  - `getFilePointer()`, `seek()`
- Reading/writing methods are very similar to various Input/OuputStreams
  - even `DataInput` and `DataOutput` are implemented

I/O as Streams



I/O and Streams

- Java provides I/O facilities in 2 packages
  - **java.io** - traditional synchronous stream-based I/O
  - **java.nio** - 'new' (a)synchronous block-based I/O
- I/O is about reading and writing data
- Reading and writing is possible using **Streams**
  - **Streams** are processed sequentially
  - **Streams** are independent of nature of data
- Java provides two types (hierarchies) of Streams
  - Byte streams (binary)
  - Character streams (unicode text)

Basic Stream processing

Reading

- open a stream (defines the source)
- while more information
  - read information
- close the stream

Provided by:

- java.io.InputStream
- java.io.Reader

Writing

- open a stream (defines the destination)
- while more information
  - write information
- close the stream

Provided by:

- java.io.OutputStream
- java.io.Writer

Stream classification

- **Direction:** input and output
- **Data type:** binary and character
- Sink streams or 'endpoints'
  - FileInputStream, ByteArrayInputStream, StringReader, etc
- Processing streams (wrappers)
  - Base classes: FilterInputStream, FilterOutputStream, FilterReader, FilterWriter
  - SequenceInputStream, ObjectInputStream, BufferedInputStream, LineNumberReader, etc
- Bridges from binary to characters
  - InputStreamReader, OutputStreamWriter

Basic operations

- Reader and InputStream
  - `int read()` - reads a single byte/character (returned as int)
  - `int read(byte/char buf[])` - reads as much bytes as possible into the passed array, returns number of bytes read
  - `int read(byte/char buf[], int offset, int length)` - the same, but works with the specified portion of an array
  - In addition, both support marking locations within a stream, skipping input data, and resetting current position
- Writer and OutputStream
  - `void write(...)` methods are analogous with reading
  - `void flush()` - flushes all buffered data to the output

Opening and Closing

- Streams are automatically opened on creation
  - If you have a stream instance - it is ready for reading or writing
- Closing is explicit
  - `close()` method closes the stream:
  - `flush()` is implicit during closing of output streams
  - Frees all underlying resources
  - Is not the same as object destruction
  - Always call it as early as possible
  - After `close()` is called, any reads or writes will fail
  - Closing several times is safe

IOException

- I/O classes throw checked *IOExceptions*
- Used by both **java.io** and **java.nio**
- There are many more specific derived exceptions, like *FileNotFoundException*, *EOFException*, *CharacterCodingException*, etc
- Even the `close()` method can throw an *IOException*

In-memory I/O

- These streams operate on in-memory data structures, which are passed to them on creation
  - *ByteArrayInputStream*, *ByteArrayOutputStream*
  - *CharArrayReader*, *CharArrayWriter*
  - *StringReader*, *StringWriter*
  - *StringBufferInputStream* (deprecated)
- Useful for mocking streams

File I/O

- Reading files
  - *FileInputStream* - reads binary files
  - *FileReader* - reads text files using the default encoding
    - *InputStreamReader* can be used for other encodings
- Writing files
  - *FileOutputStream* - writes binary files
  - *FileWriter* - writes text files using the default encoding
- Task:
  - write a simple 'copy' program (*SimpleCopyProgram* class), implement *net.azib.java.lessons.io.FileCopier*

Buffering

- These streams wrap other streams to provide buffering capabilities
  - *BufferedInputStream*, *PushbackInputStream*
  - *BufferedOutputStream*
  - *BufferedReader*, *PushbackReader*
  - *BufferedWriter*
- Task:
  - write *BufferedCopyProgram* (implementing *FileCopier*)
  - measure performance of both implementations with `System.currentTimeMillis()`

Formatted printing

- Provide convenient printing (e.g. to the console, file, another *Writer*, or *OutputStream*)
- Write-only
  - *PrintWriter* (is preferred)
  - *PrintStream* (*System.out* and *System.err*)
- Often other writable streams are wrapped into these
- They do internal buffering, either a newline char or special method invocations automatically flush the data in case *autoFlush* is enabled
- Warning: *IOExceptions* are never thrown out!
  - `checkError()` may be used for error checking

Misc features

- Concatenation:
  - *SequenceInputStream* – allows to concatenate multiple *InputStreams* together
- Pipes:
  - *PipedInputStream*, *PipedOutputStream*, *PipedReader*, *PipedWriter* – allows to connect *InputStream* to an *OutputStream*
- Counting:
  - *LineNumberReader* – counts line numbers while reading
- Peeking ahead:
  - *PushbackInputStream* and *PushbackReader* – allows to return read data back to stream
- Arbitrary data:
  - *DataInputStream*, *DataOutputStream* – allows to read/write primitive types easily

Serialization

- Java natively supports serialization of data (persistent storage of objects' internal state)
  - Serialization: *ObjectOutputStream*
  - Deserealization: *ObjectInputStream*
- Interfaces
  - *Serializable* - marker but has some methods documented
  - *Externalizable* - defines methods for saving/restoring data manually during the serialization
- It is highly recommended to define `static final long serialVersionUID` field in serializable classes (used for compatibility checks), otherwise it is computed automatically
- fields, declared as **transient** or **static**, are not serialized
- Can be used for **deep** cloning, faster than cloning recursively